COMP1730/COMP6730
Programming for Scientists

Functions: advanced topics

Academic integrity (reminder)

* Academic integrity is taken seriously at ANU! – Academic Integrity Rule 2021 is a legal document at the University
* Discussing programming problems (e.g. from labs) and ways to solve them with other students is a great way to learn – just don't discuss assessment problems
* All assignments are **individual**. You must write your own code, and be able to show that you understand every aspect of what you have written
* Suspected cheating/plagiarism will be investigated seriously accordingly to ANU academic integrity rule

Lecture outline

* Namespaces & references
* Recursion revisited

Namespaces and function calls

* Assignment statement (e.g., `variable=value`) associates a variable name with a reference to a value
* This association is stored in a **namespace** (a.k.a. **frame**), i.e., a mapping among variable names and references to values
* Whenever a function is called, a new **local namespace** is dynamically created
* Assignments to variables (including parameters) during execution of the function are registered in the local namespace
* The local namespace **disappears** when the function call ends

Scope of a variable

* The **scope** of a variable is **the set of program statements over which a variable exists**, i.e., that can refer to the variable
* In other words, the set of program statements over which the namespace the variable is defined in persists
* Because there are several namespaces, **there can be different variables with the same name in different scopes**

Different vars with same name in different scopes

```
def f(x):
    y = x ** 2
    return y - 1

x = 1
y = f(x + 1)
```

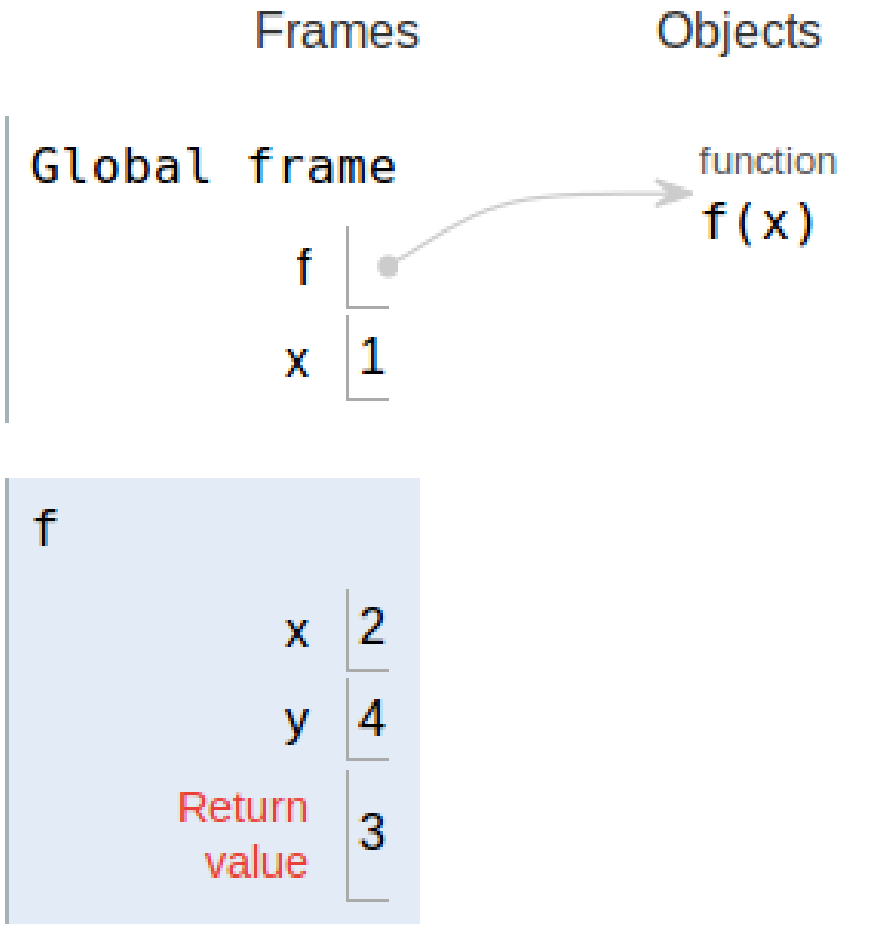


Image from `pythontutor.com`

Different vars with same name in different scopes

```
def f(x):
    y = x ** 2
    return y - 1

x = 1
y = f(x + 1)
```

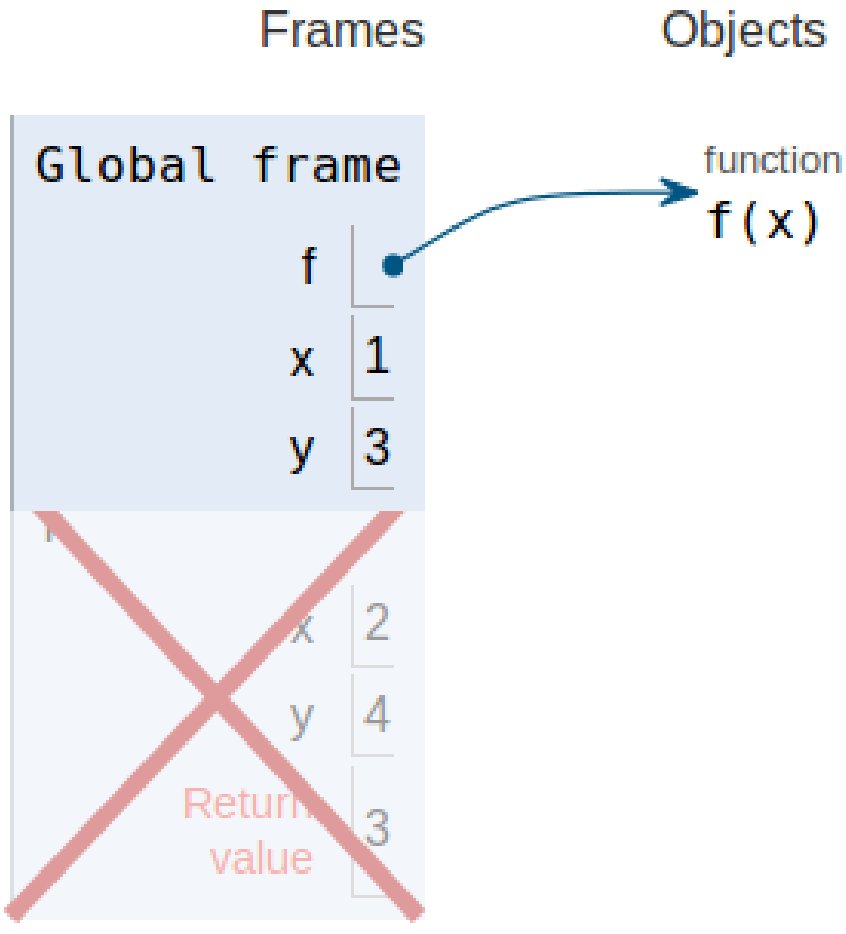


Image based on `pythontutor.com`

The local assignment rule

* Python considers a variable that is assigned **anywhere** in the function suite to be a **local variable** (this includes parameters)
* When a non-local variable is evaluated, its value is taken from the (enclosing) global namespace
* However, if a function assigns to a variable that is also defined in the global namespace, the local assignment **shadows** the non-local variable (i.e. as if the non-local variable did not exist)
* **WARNING**: If we refer to this local variable before assignment, Python will raise an `UnboundLocalError` (see next slide)

Example: function that reads non-local variable versus function that shadows non-local variable

```
def f(x):
    return x ** y

>>> y = 2
>>> f(2)
4
```

```
def f(x):
    if y < 1:
        y = 1
    return x ** y

>>> y = 2
>>> f(2)
UnboundLocalError:
 local variable 'y'
 referenced before
 assignment
```

Modifying is NOT assignment!

* Assignment changes/creates the association between a name and a reference to a value (in the current namespace)
* A modifying operation on a mutable object does NOT change any name–value association

```
def f(x):
  y = x ** 2
  f_list.append([x,y])
  return y

>>> f_list = []
>>> f(2)
4
>>> f(3)
9
>>> f_list
[[2, 4], [3, 9]]
```

Example of function that modifies non-local list

Function params hold references to args values

* When a function is called, its parameters are assigned **references** to the argument values
* If a parameter name refers to a **mutable object** (e.g., list, NumPy array, or dictionary), modifications to this object made in the function suite are visible outside the function's scope

Example of function that modifies mutable object through parameter

```
def f(ns):
  total = 0
  while len(ns) > 0:
        next = ns.pop(0)
        total = total + next
  return total

>>> a_list = [1,2,3]
>>> f(a_list)
6
>>> a_list
[]
```

Similar to previous example, HOWEVER, we can now tell from the function's signature that the function is going to access to such mutable object

Example of function that modifies mutable object through parameter

```
def f(ns):
    total = 0
    while len(ns) > 0:
        next = ns.pop(0)
        total = total + next
    return total

>>> a_list = [1,2,3]
>>> l_sum = f(a_list)
```

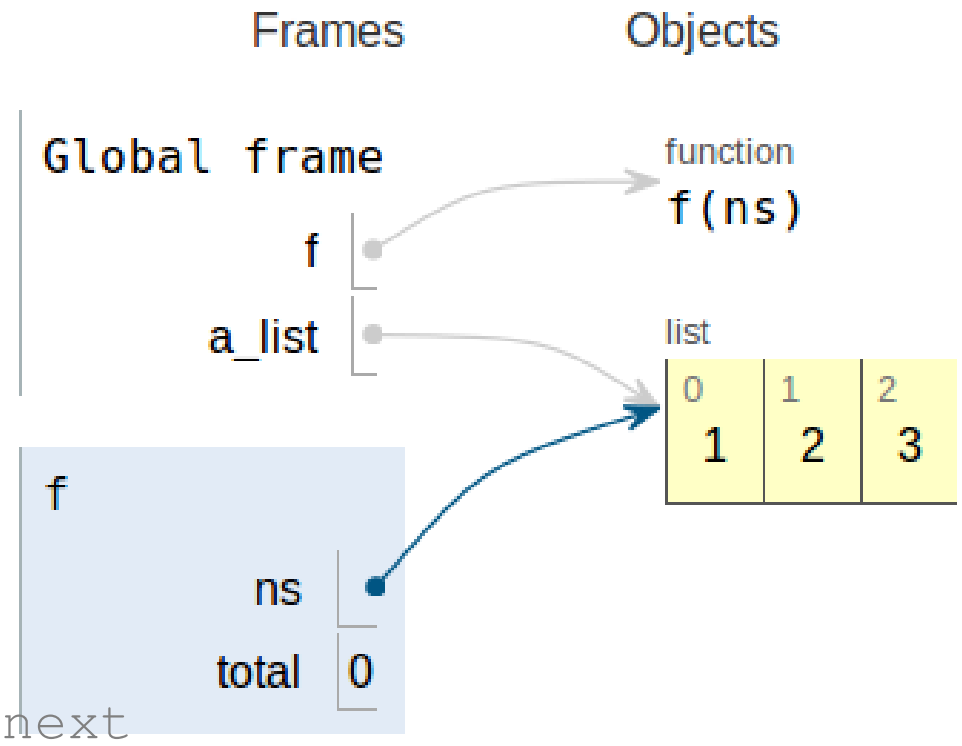


Image from pythontutor.com

Other namespaces

* Python's built-in functions (e.g., `type`, `max`, etc.) are defined in a separate namespace accessible from any part in the program
  - Programmer-defined names **override** built-in names
* Imported modules are executed in their own namespace
  - Names in a module namespace are accessed from outside by prefixing the name of the module to the name

Guidelines for good functions

* Accessing global variables within functions (specially if we modify them) is in general a bad practice that should be avoided
* Try to stick to functions that **access ONLY local variables**
  - Use parameters for all inputs to the function
  - Return all function outputs
    (for multiple outputs, return a tuple or list)
* In general, don't modify mutable argument values through the function parameters, unless there is a good reason for doing so (e.g., if it is the **specific purpose** of the function)

Recursion revisited

* A recursive function is often described as **a function that calls itself**
* Function calls form a **stack**: when the i-th function call ends, execution returns to where the call was made in the $(i-1)$-th function suite
* The function suite **MUST HAVE** a branching statement, such that a recursive call does not always take place (**base case**); otherwise, recursion never ends
* Recursion is a way to think about how to solve problems: reducing it to a smaller instance of itself

Example (contrived)

```
def f(x):
    """
      Returns 2 to the power of x
      x is an integer >= 0
    """
    if x == 0:
        return 1      # base case
    else:
        y = f(x - 1) # recursive call
    return 2 * y
```

Note that $2^x = 2 * 2^{x-1}$ for $x > 0$

```
1 def f(x):
      ...
2 y = f(2)
```

x = 2

```
3 if x == 0:
4 else:
5 y = f(x - 1)
```

x = 1

```
6 if x == 0:
7 else:
8 y = f(x - 1)
```

x = 0

```
9 if x == 0:
10 return 1
```

x = 1, y = 1

```
11 return 2 * y
```

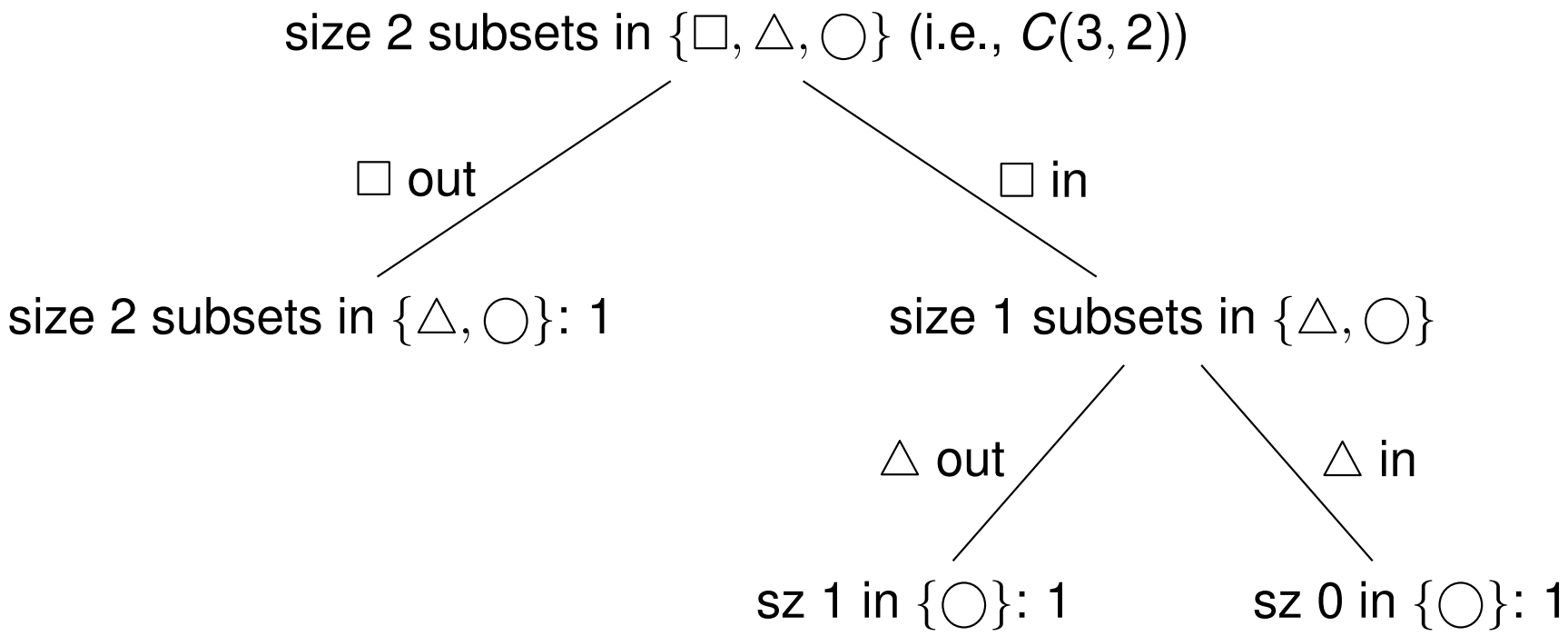
x = 2, y = 2

```
12 return 2 * y
```

y = 4

Another example

* Compute number of different subsets with k elements (i.e., of size k) in a set with n elements ($n \geq k \geq 0$)
* Denoted as $C(n, k)$ (example with $n = 3$, $k = 2$)



* Recursive formulation:

$$C(n,k) = C(n-1,k) + C(n-1,k-1)$$
$$C(n,0) = 1$$
$$C(n,n) = 1$$

```
def C(n, k):
  if k==n or k == 0:
      return 1  # base cases
  else:
      return C(n-1, k) + C(n-1, k-1) # recursive calls
```

```
1  ans = choices(3,2)
          n = 3, k = 2
2         if k == 0 or k == n:
3         else:
4             choices(n - 1, k)
                  n = 2, k = 2
5                 if k == 0 or k == n:
6                 return 1
7             choices(n - 1, k - 1)
                  n = 2, k = 1
8                 if k == 0 or k == n:
9                 else:
10                choices(n - 1, k)
                      n = 1, k = 1
11                    if k == 0 or k == n:
12                    return 1
13                choices(n - 1, k - 1)
                      n = 1, k = 0
14                    if k == 0 or k == n:
```

```
4  choices(n - 1, k)
                  n = 2, k = 2
                |  5  if k == 0 or k == n:
                |  6  return 1
7  choices(n - 1, k - 1)
                |     n = 2, k = 1
                |  8  if k == 0 or k == n:
                |  9  else:
                | 10  choices(n - 1, k)
                |                |     n = 1, k = 1
                |                | 11  if k == 0 or k == n:
                |                | 12  return 1
                | 13  choices(n - 1, k - 1)
                |                |     n = 1, k = 0
                |                | 14  if k == 0 or k == n:
                |                | 15  return 1
                | 16  return 1 + 1
17  return 1 + 2
```

ans = 3

Example: Sudoku solver